KL algorithm implementation code

Header files:

Node.h

```
#ifndef NODE_H
#define NODE_H

class Node {
      public:
         Node();
         Node(float x, float y, std::string name = "");
         Node(Node* node);
         float x();
         float y();
         std::string name();
         void setX(float x);
         void setY(float y);
         void setName(std::string name);
         void addNeighbor(Node* node);
         std::vector<Node*> neighbors();
         float distance(Node* neighbor);
      private:
         float _x;
         float _y;
         std::string _name;
         std::vector<Node*> _neighbors;
};

#endif
```

NodeKL.h

```
class NodeKL: public Node {
      public:
        NodeKL(float x, float y, std::string name = ""): Node(x, y, name) {}
        NodeKL() {}
        void setDvalue(float D);
        float getDvalue();
        std::vector<std::pair<NodeKL*,float>> neighbors();
        void addNeighbor(NodeKL* node, float distance, bool distance_update);
        void update_distance (NodeKL* node, float distance);
        int getGroup();
        void setGroup(int g);
        float is_node2_connected(NodeKL node2);
      private:
        float _Dvalue;
```

```
        int _Group;
        std::vector<std::pair<NodeKL*,float>> _neighbors;
};
```

---

read_data_helper.h

```
#include <string>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <sstream>
#include <regex>

NodeKL* read_node_names(std::string nodes_file, unsigned int * numnodes,
unsigned int * numterminals) {
    NodeKL* nodeList;
    std::string line = "";

    std::ifstream file(nodes_file.c_str());
    while (line.find("NumNodes") == std::string::npos) {
      getline(file,line);
    }
    std::regex pattern (":[ \\t]*(\\d*)");
    std::smatch match;
    bool pattern_matched = std::regex_search (line, match, pattern);
    //std::cout << match;
    //for (auto x:match) std::cout << "Contains:" << x << " ";
    //std::cout << std::endl;
    if ( pattern_matched ) {
        *numnodes = atoi(match[1].str().c_str());
        nodeList = new NodeKL[*numnodes];
    }
    //std::cout << *numnodes << std::endl;
    getline(file,line);
    pattern_matched = std::regex_search (line, match, pattern);
    if ( pattern_matched ) {
       *numterminals = atoi(match[1].str().c_str());
       //std::cout << *numterminals << std::endl;
    }
    std::string node_name;
    for ( unsigned int i = 0; i < *numnodes; i++ ) {
       getline(file,line);
       std::stringstream sstm(line);
       sstm >> node_name;
       nodeList[i].setName(node_name);
       std::cout << i << ": "<< nodeList[i].name() << std::endl;
    }
    file.close();
    return nodeList;
}

std::string get_node_type (std::string node) {
    if ( node.find("a") == std::string::npos )
       return "p";
    else
       return "a";
}

int get_node_num (std::string node) {
    return atoi(node.substr(1).c_str());
```

```
}

int read_nets(std::string nets_file, NodeKL* node_data, unsigned int num_term,
unsigned int numnodes) {
    int degree;
    int ctr = 1;
    int ctr2 = 1;
    NodeKL *curr_KL_node, *neighbor_KL_node;

    std::string curr_node, node_type, neighbor_node, neighbor_type;
    int node_num = 2, neighbor_num = 2;
    int starting_pos = nets_file.find("E") + 1;
    int num_characters = nets_file.find("_",nets_file.find("_") + 1);
    int numnets = atoi(nets_file.substr(starting_pos,num_characters).c_str());
    //std::cout << numnets;
    //Node * data [numnets];
    std::string mytext, node[numnodes+1], tmp;
    std::ifstream file(nets_file.c_str());
    //ifstream file("/home/mohit/README-FIRST.txt");
    getline(file,mytext);
    while (mytext.find("NumPins") == std::string::npos) {
        getline(file,mytext);
    }
    //cout << mytext << "\n";
    tmp = mytext.substr(mytext.find(":") + 2);
    //std::cout << tmp << std::endl;
            //std::stringstream ss(tmp);
            //ss >> numpins;
            //numpins = stoi(mytext.substr(mytext.find("NumPins") + 2));
    // int numpins;
    // numpins = atoi(tmp.c_str());

    while (ctr <= numnets) {
        getline(file,mytext);
        if ( mytext.find("NetDegree") != std::string::npos) {
            tmp = mytext.substr(mytext.find(":") + 2);
            //std::cout << tmp << " ";
            degree = atoi(tmp.c_str());
            //data[ctr]->degree = degree;
            //data[ctr]->node = "";
            //data[ctr]->next = NULL;
            //std::cout << degree << "\n";
            ctr2 = 1;
            while ( ctr2 <= degree) {
                getline(file,mytext);
                node[ctr2] = mytext.substr(0,mytext.find(" "));
                //std::cout << node[ctr2] << " " << std::endl;
                ctr2++;
            }
            ctr2 = 1;
            while ( ctr2 <= degree ) {
              //std::cout << "Current node ::" << node[ctr2] << "__" <<
std::endl;
              curr_node = node[ctr2];
              node_type = get_node_type(curr_node);
              node_num  = get_node_num(curr_node);
              //std::cout << node_type << " " << node_num << std::endl;
              if ( node_type == "a" ) {
                 curr_KL_node = &node_data[node_num + num_term];
              } else {
                 curr_KL_node = &node_data[node_num - 1];
              }
              for ( int ctr4 = ctr2 + 1; ctr4 <= degree; ctr4++ ) {
                //std::cout << "Neighbor node\t" << node[ctr4] << std::endl;
```

```
                neighbor_node = node[ctr4];
                neighbor_type = get_node_type(neighbor_node);
                neighbor_num = get_node_num(neighbor_node);
                if ( neighbor_type == "a") {
                   neighbor_KL_node = &node_data[neighbor_num + num_term];
                } else {
                   neighbor_KL_node = &node_data[neighbor_num - 1];
                }
                float dist = static_cast<float>(100 / (degree - 1)) / 100;
                //std::cout <<
"*********************************************\n";
                //std::cout << curr_KL_node->name() << "***********
Neighbors :";
                //for (std::pair<NodeKL*,float> &neighbor: curr_KL_node-
>neighbors()) {
                //   std::cout << neighbor.first->name() << " " ;
                //}
                //std::cout << std::endl;
                //std::cout << "NN:" << neighbor_KL_node->name() << "\n";
                //printf ("\n%0.2f %d\n", dist, degree);
                //std::cout <<
"*********************************************\n";
                curr_KL_node->addNeighbor(neighbor_KL_node, dist, true);
              }
              ctr2++;
            }
            ctr++;
        }
    }

    file.close();
    return 0;
}

void display_data(NodeKL *node_data, unsigned int numnodes) {
     for (unsigned int i = 0; i < numnodes; i++) {
       std::cout << node_data[i].name() << std::endl;
        for (std::pair<NodeKL*, float> &neighbor: node_data[i].neighbors()) {
          std::cout << neighbor.first->name() << " : " << neighbor.second << "
";
       }
       std::cout << std::endl;
     }

}
```

KL_helper.h

```
#include <iostream>

#include <string>
#include <random>
#include <set>
#include <algorithm>

void show_partitions(std::vector<NodeKL*> &group1, std::vector<NodeKL*> &group2,
unsigned int numnodes) {
    std::cout << "\033[1;36mGroup1 : \033[0m";
```

```
     for ( unsigned int i = 0; i < (numnodes/2); i++)
       std::cout << group1[i]->name() << " ";
    std::cout << std::endl;
    std::cout << "\033[1;36mGroup2 : \033[0m";
    for ( unsigned int i = 0; i < (numnodes - (numnodes/2)); i++)
       std::cout << group2[i]->name() << " ";
    std::cout << std::endl;
}

void cut_cost(unsigned int num_nodes, std::vector<NodeKL*> &group1, bool final =
false) {
    int group, connecting_node_group;
    float cost=0, connecting_node_distance;
    std::string connecting_node, node;
    //std::cout << "\n\nCalculating cut cost for the current partition.....\n";
    for( unsigned int i=0;i<(num_nodes/2);i++) {
       node = (*group1[i]).name();
       //std::cout << "Node : " << node << "\n";
       group = 1;
       //std::cout << "Group : " << group << "\n";
       for (std::pair<NodeKL*, float> &neighbor: (*group1[i]).neighbors()) {
         connecting_node = neighbor.first->name();
         //std::cout << "Connecting node : " << connecting_node << "\n";
         connecting_node_group = neighbor.first->getGroup();
         connecting_node_distance = neighbor.second;
         //std::cout << "Connecting node group : " << connecting_node_group <<
"\n";
         if ( group != connecting_node_group ) {
            cost+=connecting_node_distance;
         }
       }
       //std::cout << "**********************************\n";
    }
    if ( final ) {
       std::cout << "\n\n******Cut cost of final partitioning : \033[1;31m" <<
cost << "\033[0m ********\n\n";
    } else {
       std::cout << "\n\nCut cost of current partitioning : \033[1;31m" << cost
<< "\033[0m\n\n";
    }

}

void initial_partition(unsigned int num_nodes, std::vector<NodeKL*> &group1 ,
std::vector<NodeKL*> &group2, NodeKL * node_data) {
  unsigned int i = 1;
  int j = 0;
  std::set<int> numbers,numbers2;
  std::set<int> complete_set;

  std::cout << "\n\n Making initial bi-partition..... \n" << num_nodes;
  for ( i = 0; i < num_nodes ; i++) {complete_set.insert(j);j++;}
  srand( (unsigned)time(NULL));
  while (numbers.size() < (num_nodes / 2 )) {
    numbers.insert((rand() % num_nodes));
  }
  //numbers.insert(0);
  //numbers.insert(1);
  //numbers.insert(2);
  //numbers.insert(8);
  //numbers.insert(9);
  std::set_difference(complete_set.begin(), complete_set.end(), numbers.begin(),
numbers.end(), inserter(numbers2, numbers2.end()));
  for (auto const& e : numbers) {
```

```
    group1.push_back(&node_data[e]);
    node_data[e].setGroup(1);
  }
  for (auto const& e : numbers2) {
    group2.push_back(&node_data[e]);
    node_data[e].setGroup(2);
  }
  show_partitions(group1, group2, num_nodes);
}

void initial_d_values(unsigned int num_nodes, NodeKL * data) {
    int group, connecting_node_group;
    std::string connecting_node;
    float connecting_node_distance;
    //std::cout << "\n\nCalculating initial D values .....\n";
    for(unsigned int i=0;i<num_nodes;i++) {
       data[i].setDvalue(0);
       //std::cout << "Node : " << data[i].name() << "\n";
       group = data[i].getGroup();
       //std::cout << "Group : " << group << "\n";
       for (std::pair<NodeKL*, float> &neighbor: data[i].neighbors()) {
         connecting_node = neighbor.first->name();
         //std::cout << "Connecting node : " << connecting_node << "\n";
         connecting_node_group = neighbor.first->getGroup();
         connecting_node_distance = neighbor.second;
         //std::cout << "Connecting node group : " << connecting_node_group <<
"\n";
         if ( group == connecting_node_group ) {
            data[i].setDvalue(data[i].getDvalue() - connecting_node_distance);
         } else {
            data[i].setDvalue(data[i].getDvalue() + connecting_node_distance);
         }
       }
       //std::cout << "D value of node " << data[i].name() << ": " <<
data[i].getDvalue() << "\n";
       //std::cout << "************************************\n";
    }
    std::cout << "************************************\n";

}

void d_values_stats(unsigned int num_nodes, NodeKL * data) {
    int positive_d_values = 0, negative_d_values = 0, zero_d_values = 0, total =
0;
    for(unsigned int i=0;i<num_nodes;i++) {
       if ( data[i].getDvalue() > 0 )
          positive_d_values++;
       if ( data[i].getDvalue() < 0 )
          negative_d_values++;
       if ( data[i].getDvalue() == 0 )
          zero_d_values++;
    }
    total = positive_d_values + negative_d_values + zero_d_values;
    std::cout << positive_d_values << "::" << negative_d_values << "::" <<
zero_d_values << "::" << total << "\n";
}
```

KL_graphHelper.h

```
// Taken from https://github.com/abangfarhan/graph-
sfml/blob/master/include/graphHelper.h


#include <math.h>
#include <time.h>

#define PI 3.14159265

sf::RectangleShape Line(float x1, float y1, float x2, float y2, float thickness
= 1) {
     sf::RectangleShape line;
     float len, angle;

     line.setPosition(x1, y1);
     len = sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
     angle = atan((y2 - y1) / (x2 -x1)) * 180 / PI;
     // line pointing down-left and top-left must be incremented by 180 deg
     if ( x2 - x1 < 0 ) angle += 180;

     line.setSize(sf::Vector2f(len, thickness));
     line.setFillColor(sf::Color(100, 100, 100));
     line.setRotation(angle);
     return line;
}

void set_xy_coordinates_for_nodes( unsigned int numnodes, NodeKL * data, int
max_x, int max_y ) {
     float x, y, tmp;
     srand(time(NULL));
     for (unsigned int i = 0; i < numnodes; ++i) {
       switch ( i % 10 ) {
         case 0: tmp = (max_x / 22);
                 break;
         case 1: tmp = (2 * max_x / 22);
                 break;
         case 2: tmp = (3 * max_x / 22);
                 break;
         case 3: tmp = (4 * max_x / 22);
                 break;
         case 4: tmp = (5 * max_x / 22);
                 break;
         case 5: tmp = (6 * max_x / 22);
                 break;
         case 6: tmp = (7 * max_x / 22);
                 break;
         case 7: tmp = (8 * max_x / 22);
                 break;
         case 8: tmp = (9 * max_x / 22);
                 break;
         case 9: tmp = (10 * max_x / 22);
                 break;
       }
       if ( data[i].getGroup() == 1 ) {
          x = tmp;
       } else {
          x = (max_x / 2) + tmp;
       }
       y = rand() % max_y;
```

```
        data[i].setX(x);
        data[i].setY(y);
      }
}

void drawGraph(unsigned int n_nodes, NodeKL * nodeList, int height, int width) {
      sf::CircleShape nodeCircles[n_nodes];
      for (unsigned int i = 0; i < n_nodes; ++i) {
         float radius = 5;
         nodeCircles[i].setRadius(radius);
         nodeCircles[i].setPosition(sf::Vector2f(nodeList[i].x() - radius,
nodeList[i].y() - radius));
         nodeCircles[i].setFillColor(sf::Color(255, 0, 0, 90));
      }
      sf::ContextSettings settings;
      settings.antialiasingLevel = 8;
      sf::RenderWindow window(sf::VideoMode(width, height), "Graph",
sf::Style::Default, settings);
      while (window.isOpen()) {
       sf::Event event;
       while (window.pollEvent(event)) {
         if ( event.type == sf::Event::Closed) {
            window.close();
         }
       }
       window.clear(sf::Color::White);
       for (unsigned int i = 0; i < n_nodes; ++i) {
           window.draw(nodeCircles[i]);
           for (std::pair<NodeKL*,float> &neighbor: nodeList[i].neighbors()) {
               sf::RectangleShape line = Line(nodeList[i].x(), nodeList[i].y(),
neighbor.first->x(), neighbor.first->y(), neighbor.second);
               if (neighbor.second > 0.67 ) {
                  line.setOutlineColor(sf::Color(255,0,0,200));
                  line.setFillColor(sf::Color(255,0,0));
               } else if ( neighbor.second > 0.33 ) {
                  line.setOutlineColor(sf::Color(0,255,0,200));
                  line.setFillColor(sf::Color(0,255,0));
               } else {
                  line.setOutlineColor(sf::Color(0,0,255,200));
                  line.setFillColor(sf::Color(0,0,255));
               }
               window.draw(line);
           }
       }
       float dividing_line_thickness = 5;
       sf::RectangleShape line = Line((width/2),0,
(width/2),height,dividing_line_thickness);
       line.setFillColor(sf::Color(0,0,255));
       line.setOutlineColor(sf::Color(0,0,255));
       window.draw(line);
       //sf::RectangleShape rect1(sf::Vector2f((width/2)-4, height-4));
       //rect1.setFillColor(sf::Color(255,255,255,10));
       //rect1.setPosition(2,2);
       //rect1.setOutlineThickness(2);
       //rect1.setOutlineColor(sf::Color(0,0,255));
       //window.draw(rect1);
       //sf::RectangleShape rect2(sf::Vector2f((width/2), height));
       //rect2.setPosition((width/2) + 1, 0);
       //rect2.setFillColor(sf::Color(255,255,255,10));
       //window.draw(rect2);
       window.display();
    }
}
```

KL_bipartition_main.h

```
#include <limits>
#include <iostream>
#include <vector>
#include "KL_graphHelper.h"

void get_largest_decrease_cutcost_exchange_nodes(std::vector<NodeKL*> A,
std::vector<NodeKL*> B, float * max_gain, NodeKL ** ELE1, NodeKL ** ELE2,
unsigned int num_nodes) {
       NodeKL node1, node2;
       float gain;
       *max_gain = INT_MIN;
       for (unsigned int i=0; i<A.size(); i++) {
         node1 = *A[i];
         //std::cout << "\nGroup1 element, node1 = " << node1.name() << "\n";
         for (unsigned int j=0; j<B.size(); j++) {
           node2 = *B[j];
           //std::cout << "\nGroup2 element, node2 = " << node2.name() << "\n";
           gain = node1.getDvalue() + node2.getDvalue() -
2.0*(node1.is_node2_connected(node2));
           gain = std::ceil(gain * 100.0) / 100.0;
           if ( gain > *max_gain ) {
              *max_gain = gain; *ELE1 = A[i]; *ELE2 = B[j];
           }
           //std::cout << "Node1 " << node1.name() << " : Node1 d value" <<
node1.getDvalue() << " :: " << " Node2 " << node2.name() << " : Node2 d value"
<< node2.getDvalue() << " : Node 1 node 2 connected : " <<
node1.is_node2_connected(node2) << " :: " << "Gain " << gain << "\n";
           //std::cout <<  node1.name() << ":" << node1.getDvalue() << "::" <<
node2.name() << ":" << node2.getDvalue() << "::" <<
node1.is_node2_connected(node2) << "::" << gain << "__________";
         }
         //std::cout << "\n";
       }
       //std::cout << "\n\nExiting
get_largest_decrease_cutcost_exchange_nodes...\n\n";

}

void delete_element(std::vector<NodeKL*> *group, NodeKL ele) {
     for (std::vector<NodeKL*>::iterator it=group->begin(); it != group->end();
) {
          if ( (*it)->name() == ele.name() ) {
             it = group->erase(it);
          } else {
             ++it;
          }
     }
}

void swap_element_for_group(std::vector<NodeKL*> *group, NodeKL *ele1, NodeKL
*ele2) {
     for (std::vector<NodeKL*>::iterator it=group->begin(); it != group->end();
++it) {
          if ( (*it)->name() == ele1->name() ) {
             (*it) = ele2;
          }
     }
}
```

```
int update_d_values2(unsigned int num_nodes, NodeKL * data, NodeKL* node1,
NodeKL* node2) {
    std::set<NodeKL*> neighbors_of_exchange_nodes;
    int num_d_values_updated = 0;
    for ( std::pair<NodeKL*,float> &neighbor: node1->neighbors()) {
        neighbors_of_exchange_nodes.insert(neighbor.first);
    }
    for ( std::pair<NodeKL*,float> &neighbor: node2->neighbors()) {
        neighbors_of_exchange_nodes.insert(neighbor.first);
    }
    for (auto const& e : neighbors_of_exchange_nodes) {
        //std::cout << e->name() << ":" << e->getDvalue() << "::";
        if (e->getGroup() == node1->getGroup()) {
           e->setDvalue(e->getDvalue() + 2 * e->is_node2_connected(*node1) - 2 *
e->is_node2_connected(*node2));
                num_d_values_updated++;
        } else {
           e->setDvalue(e->getDvalue() + 2 * e->is_node2_connected(*node2) - 2 *
e->is_node2_connected(*node1));
                num_d_values_updated++;
        }
        //std::cout << e->getDvalue() << "\n";
    }
    //std::cout << "\n";
    return num_d_values_updated;
}

int update_d_values_(unsigned int num_nodes, NodeKL * data, NodeKL* node1,
NodeKL* node2) {
     int num_d_values_updated = 0;
     for ( std::pair<NodeKL*,float> &neighbor: node1->neighbors()) {
         if (neighbor.first->getGroup() == node1->getGroup()) {
            if (neighbor.first->getDvalue() != (neighbor.first->getDvalue() + 2
- 2 * (neighbor.first->is_node2_connected(*node2))))
                num_d_values_updated++;
            neighbor.first->setDvalue(neighbor.first->getDvalue() + 2 - 2 *
(neighbor.first->is_node2_connected(*node2)));
         } else {
            if (neighbor.first->getDvalue() != (neighbor.first->getDvalue() + 2
* (neighbor.first->is_node2_connected(*node2)) - 2))
                num_d_values_updated++;
            neighbor.first->setDvalue(neighbor.first->getDvalue() + 2 *
(neighbor.first->is_node2_connected(*node2)) - 2);
         }
     }
     for ( std::pair<NodeKL*,float> &neighbor: node2->neighbors()) {
         if (neighbor.first->getGroup() == node2->getGroup()) {
            if (neighbor.first->getDvalue() != (neighbor.first->getDvalue() + 2
- 2 * (neighbor.first->is_node2_connected(*node1))))
                num_d_values_updated++;
            neighbor.first->setDvalue(neighbor.first->getDvalue() + 2 - 2 *
(neighbor.first->is_node2_connected(*node1)));
         } else {
            if (neighbor.first->getDvalue() != (neighbor.first->getDvalue() + 2
* (neighbor.first->is_node2_connected(*node1)) - 2))
                num_d_values_updated++;
            neighbor.first->setDvalue(neighbor.first->getDvalue() + 2 *
(neighbor.first->is_node2_connected(*node1)) - 2);
         }
     }
     return num_d_values_updated;
}
```

```
void update_d_values(unsigned int num_nodes, NodeKL * data, NodeKL node1, NodeKL
node2) {
     NodeKL same_group_node, different_group_node;
     int group;
     //std::cout << "\n\nUpdating D values ........\n";
     for (unsigned int i = 0; i < num_nodes; i++) {
        group = data[i].getGroup();
        if ( group == 1 ) {
           same_group_node = node1; different_group_node = node2;
        } else {
           same_group_node = node2; different_group_node = node1;
        }
        data[i].setDvalue(data[i].getDvalue() + 2 *
data[i].is_node2_connected(same_group_node) -  2 *
data[i].is_node2_connected(different_group_node));
        //std::cout << "D value of " << data[i].name() << ": " <<
data[i].getDvalue() << "\n";
        //std::cout << data[i].name() << ": " << data[i].getDvalue() << "__";
     }
     //std::cout << "\n\n";

}

int max_element_index(float * g, unsigned int max_index) {
    float max = -FLT_MAX;
    unsigned int index, index_of_max;
    for (index=0;index < max_index;index++) {
        if ( g[index] > max ) {
           max = g[index];
           index_of_max = index;
        }
    }
    return index_of_max;
}

void swap_ele(std::vector<NodeKL*> * G1, std::vector<NodeKL*> * G2, NodeKL
*ele_A, NodeKL *ele_B) {
     //std::vector<NodeKL*>::iterator it;
     //std::cout << "\nEnter swap\n";
     //std::cout << "Before swap:: G1: ";
     //for (it = G1->begin(); it != G1->end(); ++it ) {
     //    std::cout << (*it)->name() << " ";
     //}
     //std::cout <<  ":: G2: ";
     //for (it = G2->begin(); it != G2->end(); ++it ) {
     //    std::cout << (*it)->name() << " ";
     //}
     //std::cout << "\n\n";
     ele_B->setGroup(1);
     swap_element_for_group(G1, ele_A, ele_B);
     //std::cout << "\nAfter delete A\n";
     //std::cout << "\nAfter append A\n";
     ele_A->setGroup(2);
     swap_element_for_group(G2, ele_B, ele_A);
     //std::cout << "\nAfter delete B\n";
     //std::cout << "\nAfter append B\n";
     //std::cout << "After swap:: G1: ";
     //for (it = G1->begin(); it != G1->end(); ++it ) {
     //    std::cout << (*it)->name() << " ";
     //}
     //std::cout <<  ":: G2: ";
     //for (it = G2->begin(); it != G2->end(); ++it ) {
     //    std::cout << (*it)->name() << " ";
     //}
```

```
     //std::cout << "\n\n*************************************************\n\n";
}

void main_pass(unsigned int num_nodes, NodeKL * data, std::vector<NodeKL*> &A,
std::vector<NodeKL*> &B, int width, int height) {
     int j,k, iteration = 1, color=0;
     float actual_gain = FLT_MAX, max_gain, g[num_nodes/2],Gain[num_nodes/2];
     bool colored_print = false;
     NodeKL *max_gain_ele1, *max_gain_ele2, *to_swap_ele_A[num_nodes/2],
*to_swap_ele_B[num_nodes/2];
     //std::cout << A.size() << "\n";
     //std::cout << sizeof(NodeKL) << "\n";
     std::vector<NodeKL*> group1 (A);
     std::vector<NodeKL*> group2 (B.begin(), B.end());
     max_gain_ele1 = group1[0];
     max_gain_ele2 = group2[0];
     while (actual_gain > 0 ) {
        std::cout << "\n\nPass : " << iteration << "\n\n";
        std::cout << "\n\n\nPartitions before pass are:\nA:";
        for ( auto it = group1.begin(); it != group1.end(); ++it ) {
            std::cout << (*it)->name() << " ";
        }
        std::cout << "\nB:";
        for ( auto it = group2.begin(); it != group2.end(); ++it ) {
            std::cout << (*it)->name() << " ";
        }
        cut_cost(num_nodes,group1);
        initial_d_values(num_nodes,data);
        for (unsigned int i=0;i<num_nodes/2;i++) {

get_largest_decrease_cutcost_exchange_nodes(group1,group2,&max_gain,&max_gain_el
e1,&max_gain_ele2,num_nodes);
            g[i] = max_gain; to_swap_ele_A[i] = max_gain_ele1; to_swap_ele_B[i]
= max_gain_ele2;
            //std::cout << "\n\nMax gain achieved in iteration " << i << " is:"
<< max_gain << "\n";
            //std::cout <<
"*****************************************************************\n";
            delete_element(&group1, *max_gain_ele1);
            delete_element(&group2, *max_gain_ele2);
            //std::cout << "\n\nA after deletion of " << max_gain_ele1->name()
<< " is: ";
            //for ( auto it = group1.begin(); it != group1.end(); ++it ) {
            //    std::cout << (*it)->name() << " ";
            //}
            //std::cout << " :: B after deletion of " << max_gain_ele2->name()
<< " is: ";
            //for ( auto it = group2.begin(); it != group2.end(); ++it ) {
            //    std::cout << (*it)->name() << " ";
            //}
            //std::cout  << "\n\n";
            //std::cout << "\n\nCheck\n";
            //std::cout << "\nLength of A:" << A.length() << "\n\n";
            //std::cout << "Exchange nodes are : " << max_gain_ele1->name() <<
"::" << max_gain_ele2->name() << "\n";
            if ( group1.size() > 0 ) {
               update_d_values2(num_nodes,data,max_gain_ele1,max_gain_ele2);
            }
            //d_values_stats(num_nodes, data);
        }
        Gain[0] = g[0];
        //std::cout << "Gain[0] = " << Gain[0] << "\n";
        for (unsigned int i=1;i<num_nodes/2;i++) {
            //std::cout << i << "\n";
```

```
            Gain[i] = Gain[i-1] + g[i];
            //std::cout << "Gain[" << i << "] = " << Gain[i] << "\n";
        }
        //std::cout << "\n\nCheck\n\n";
        //k = distance(Gain, max_element(Gain, Gain + num_nodes/2));
        k = max_element_index(Gain,num_nodes/2) + 1;
        actual_gain = Gain[k-1];
        if ( actual_gain - 0.01 <= 0 ) {
           actual_gain = 0;
        }
        if ( actual_gain > 0 ) {
           std::cout << "\n\nNumber of elements to swap (k value) : " << k <<
"\n\n";
           for (j=0;j<k;j++) {
               swap_ele(&A, &B, to_swap_ele_A[j], to_swap_ele_B[j]);
           }
        }
        group1 = A; group2 = B;
        std::cout << "\n\n\nPartitions after pass are(Swapped elements
highlighted):\nA:";
        for ( auto it = group1.begin(); it != group1.end(); ++it ) {
            colored_print = false;
            for (j=0;j<k;j++)
               if ( (*it)->name() == to_swap_ele_B[j]->name() ) {
                  colored_print = true;
                  color = j;
               }
            if ( colored_print ) {
               std::cout << "\033[38;5;" << color << "m\"" << (*it)->name() <<
"\"\033[0m ";
            } else {
               std::cout << (*it)->name() << " ";
            }
        }
        std::cout << "\nB:";
        for ( auto it = group2.begin(); it != group2.end(); ++it ) {
            colored_print = false;
            for (j=0;j<k;j++)
               if ( (*it)->name() == to_swap_ele_A[j]->name() ) {
                  colored_print = true;
                  color = j;
               }
            if ( colored_print ) {
               std::cout << "\033[38;5;" << color << "m\"" << (*it)->name() <<
"\"\033[0m ";
            } else {
               std::cout << (*it)->name() << " ";
            }
        }
        std::cout << "\nGain achieved = " << actual_gain << "\n\n";
        //exit(0);
        //if ( actual_gain > 0 ) {
        //   cut_cost(num_nodes,group1,data);
        //   initial_d_values(num_nodes,data);
        //}
        //if ( actual_gain < 1 ) {
        //   set_xy_coordinates_for_nodes(num_nodes,data,width,height);
        //   drawGraph(num_nodes,data,height,width);
        //}
        iteration++;
    }
    cut_cost(num_nodes,group1,true);
    set_xy_coordinates_for_nodes(num_nodes,data,width,height);
    drawGraph(num_nodes,data,height,width);
```

```
}
```

SRC files

Node.cpp

```
// Taken from https://github.com/abangfarhan/graph-sfml/blob/master/src/Node.cpp


#include <vector>
#include <string>
#include <math.h>

#include "Node.h"

Node::Node() {
    setX(0);
    setY(0);
    setName("");
}

Node::Node(float x, float y, std::string name) {
    setX(x);
    setY(y);
    setName(name);
}

Node::Node(Node* node) {
    /* copy properties, except the neighbors */
    setX(node->x());
    setY(node->y());
    setName(node->name());
}

float Node::x() {
      return _x;
}

float Node::y() {
      return _y;
}

std::string Node::name() {
      return _name;
}

void Node::setX(float x) {
      _x = x;
}
```

```
void Node::setY(float y) {
      _y = y;
}

void Node::setName(std::string name) {
      _name = name;
}

void Node::addNeighbor(Node* node) {
     // Add neighbor node to this->_neighbors if not exist,
     // and add this to the neighbor node->_neighbors
     for (Node* neighbor: _neighbors)
         if (neighbor == node)
            return;
     _neighbors.push_back(node);
     node->addNeighbor(this);
}

std::vector<Node*> Node::neighbors() {
     return _neighbors;
}

float Node::distance(Node* neighbor) {
      return sqrt(pow(_x - neighbor->x(), 2) + pow(_y - neighbor->y(), 2));
}
```

NodeKL.cpp

```
#include <vector>
#include <string>
#include <utility>
#include <iostream>
#include <iterator>

#include "Node.h"
#include "NodeKL.h"

float NodeKL::getDvalue() {
      return _Dvalue;
}

void NodeKL::setDvalue(float D) {
     _Dvalue = D;
}


std::vector<std::pair<NodeKL*, float>> NodeKL::neighbors() {
     return _neighbors;
     //std::vector<NodeKL*> neighbors_;
     //for ( std::vector<std::pair<NodeKL*, float>>::iterator it =
_neighbors.begin(); it != _neighbors.end(); ++it) {
     //  neighbors_.push_back(it->first);
       //std::cout << "\n" << this->name() << " Current nodes neighbors: " <<
neighbor.first->name() << " ";
     //}
     //std::cout << std::endl;
     //return neighbors_;
```

```
}


void NodeKL::addNeighbor(NodeKL* node, float distance, bool distance_update) {
     //std::cout << " Current node: " << this->name() << std::endl;
     //std::cout << "Neighbor to add: " << node->name() << std::endl;
     //std::cout << "Neighbor to add distance is: " << distance << std::endl;
     for ( std::vector<std::pair<NodeKL*, float>>::iterator it =
_neighbors.begin(); it != _neighbors.end(); ++it) {
         //std::cout << "Already neighbor: " << it->first->name() << std::endl;
         //std::cout << "Already neighbor distance: " << it->second <<
std::endl;
         if (it->first->name() == node->name() ) {
            if ( distance_update ) {
               it->second += distance;
               node->update_distance(this, distance);
            }
            /*for ( std::pair<NodeKL*, float> &neighbor: node->neighbors()) {
if (neighbor.first->name() == this->name() ) {
neighbor.second +=distance;
}
}*/
            return;
         }
     }
     //std::cout << "check\n";
     _neighbors.push_back(std::make_pair(node,distance));
     node->addNeighbor(this,distance,false);
}

void NodeKL::update_distance (NodeKL* node, float distance) {
     for ( std::pair<NodeKL*, float> &neighbor: _neighbors)
         if (neighbor.first == node)
            neighbor.second += distance;
}

int NodeKL::getGroup() {
     return _Group;
}

void NodeKL::setGroup(int g) {
    _Group = g;
}

float NodeKL::is_node2_connected(NodeKL node2) {
    for (std::pair<NodeKL*,float> &neighbor: _neighbors) {
          if (neighbor.first->name() == node2.name()  ) {
             return neighbor.second;
          }
    }
    return 0.0;
}
```

KL_partitioning.cpp

```
#include <string>
#include <iostream>
#include <fstream>
```

```
#include <stdlib.h>
#include <sstream>
#include <regex>
#include <set>
#include <algorithm>
#include <climits>
#include <cfloat>
#include <typeinfo>
#include <cmath>
#include <SFML/Graphics.hpp>


#include "Node.h"
#include "NodeKL.h"
#include "read_data_helper.h"
#include "KL_helper.h"
#include "KL_bipartion_main.h"

int main(int argc, char * argv[]) {
    const int width = 1200;
    const int height = 600;
    std::string nodes_file, nets_file;
    unsigned int numnodes, numterminals;
    NodeKL* node_data;

    if ( argc > 1 ) {
        nodes_file = argv[1];
        std::cout << nodes_file;
        nets_file = argv[2];
        std::cout << nets_file;
    } else {
        nodes_file = "spp_N199_E232_R11_154.nodes.txt";
        nets_file = "spp_N199_E232_R11_154.nets.txt";
    }

    node_data = read_node_names(nodes_file, &numnodes, &numterminals);
    read_nets(nets_file, node_data, numterminals, numnodes);
    //display_data(node_data, numnodes);

    std::vector<NodeKL*> partition1;
    std::vector<NodeKL*> partition2;
    //std::cout << numnodes - (numnodes/2);
    initial_partition(numnodes, partition1, partition2, node_data);
    cut_cost(numnodes, partition1);
    initial_d_values(numnodes, node_data);

    set_xy_coordinates_for_nodes(numnodes,node_data,width,height);
    drawGraph(numnodes,node_data,height,width);
    d_values_stats(numnodes, node_data);
    //std::cout << "\n\n\n\n\n**********************************" <<
partition1[0]->getDvalue() << "******************\n\n\n\n\n";
    main_pass(numnodes, node_data, partition1, partition2,width,height);
    return 0;
}
```